Contextual Service Loading by Dependency Graph Colouring

Amira Ben Hamida
ARES INRIA / CITI,
INSA-Lyon, F-69621, France
amira.ben-hamida@insa-lyon.fr

Frédéric Le Mouël
ARES INRIA / CITI,
INSA-Lyon, F-69621, France
frederic.le-mouel@insa-lyon.fr

Stéphane Frénot
ARES INRIA / CITI,
INSA-Lyon, F-69621, France
stephane.frenot@insa-lyon.fr

Mohamed Ben Ahmed
RIADI GDL / ENSI Manouba,
2010,Tunisie
mohamed.benAhmed@riadi.rnu.tn

ABSTRACT

While installing applications on mobiles devices, we may face issues due to the limit of the device resources. In this paper, we propose AxSeL: A conteXtual Service Loader that considers services-oriented applications and loads them from a distant repository. Services dependencies are represented in a graph that is coloured considering the devices and services constraints. The graph colouring aims to take services loading decisions.

Categories and Subject Descriptors

Middleware and Operating Systems []

General Terms

Algorithms

Keywords

Service loading, middleware, context, graph theory.

1. INTRODUCTION

Les futurs environnements pervasifs proposent de fournir aux utilisateurs la possibilité d'accéder, à travers leurs dispositifs mobiles, aux applications de l'environnement immédiat et distant. Installer une application sur un dispositif mobile revient à la découvrir, la rapatrier localement, la déployer et enfin à l'exécuter. Or, les capacités d'accueil des noeuds mobiles sont très différentes. Une application pouvant être chargée sur un noeud mobile donné peut ne pas forcément l'être sur un autre. Il est donc nécessaire d'adapter celle-ci en tenant compte des contraintes du dispositif. L'adaptation permet de fournir à un plus large éventail de dispositifs la possibilité de charger les applications proposées.

Une solution est de décomposer ces applications en plusieurs modules et de ne choisir et télécharger ensuite que les parties convenant aux ressources proposées. Les approches orientées services fournissent la possibilité de concevoir les applications comme étant un ensemble de services dépendants.

Dans cet article, nous présentons AxSeL une architecture de chargement contextuel de services. Elle considère des applications orientées services que nous supposons composées de services prioritaires et d'autres moins prioritaires. Le paradigme service (cf 3.1) donne la possibilité de décrire, publier, découvrir et réutiliser les services. Les services et les noeuds mobiles possèdent des contextes de description : capacité d'accueil, taille mémoire, priorité. Lors du chargement, AxSeL procède à la confrontation des deux contextes : celui fourni par le dispositif (capacité d'accueil) et celui requis par le service (taille mémoire et priorité). Celle-ci aboutit au chargement ou non d'un service. AxSeL propose :

- Une vue unifiée des services et composants fournis, AxSeL propose une prise en compte du contexte au niveau des descripteurs de dépôts de services. Et ce à travers l'agrégation de plusieurs descripteurs dans un seul fournissant ainsi une vue unifiée des différents dépôts de services.

- Une décision contextuelle du chargement : Les services, composants et leurs dépendances sont représentés sous la forme d'un graphe (cf 3.3.1). Nous procédons ensuite à la coloration de ce graphe en tenant compte des données du service ainsi que celles de la plate-forme hôte. Deux couleurs sont utilisées pour exprimer la décision à prendre pour chaque service. Ainsi, AxSeL prend compte du contexte. Cette décision est prise dynamiquement et peut être remise en question.

Dans la suite de cet article, nous présentons l'état de l'art en section 2. Ensuite, dans la section 3, nous détaillons l'architecture d'AxSeL. La réalisation est décrite dans la section 4. Enfin, nous concluons dans la section 5.

2. ÉTAT DE L'ART

Nous décomposons notre état de l'art en deux parties : les mécanismes de chargement existant et les architectures contextuelles pour le déploiement de services.

2.1 Mécanismes de chargement

Java passe par un mécanisme de chargeur de classes pour exécuter les applications. Une classe contenant le bytecode provient soit localement du classpath standard soit d'une localisation distante. Dans une approche orientée composants comme OSGi [4], un bundle contient un ensemble de classes mises à disposition par une même unité de chargement. Dans cette approche le bundle est explicitement chargé par la passerelle et s'il ne possède pas certaines classes desquelles il dépend, il ne peut être démarré. Les dépôts de composants comme OBR [7] permettent de tracer un graphe de dépendances entre les bundles afin de garantir, lors de l'installation d'un bundle, l'installation des bundles dont il dépend. Dans une approche plus généraliste orientée services, nous partons du principe que plusieurs bundles peuvent satisfaire une dépendance. Le dépôt de bundles standard ne permet pas la sélection d'un bundle en fonction d'un autre critère que celui des dépendances de packages. AxSeL propose d'intégrer une notion de contexte permettant d'exprimer de nouveaux types de dépendances entre bundles. Cette notion de contexte permet alors de sélectionner un composant parmi plusieurs, mais permet également lors d'un changement de son état de remettre en cause certains composants déjà installés. La section suivante présente quelques approches prenant en compte le contexte.

2.2 Architectures contextuelles pour le déploiement de services

Poladian [9] propose une technique d'adaptabilité dynamique de services à l'exécution dans un environnement pervasif. Un algorithme décisionnel qui prend plusieurs paramètres contextuels tels que les préférences utilisateurs et les contraintes matérielles, y est présenté. Il n'y est cependant pas question de gestion de dépendances de services. CASM [8] est une architecture pour le développement et le prototypage des services conscients du contexte. Les informations contextuelles sont collectées et interprétées pour fournir aux services des tâches d'exécution. La contextualisation peut aussi se faire par un changement du comportement des services. SOCAM [3] est une architecture qui permet comme CASM de construire et de prototyper des services mobiles conscients du contexte. Un ensemble de règles prédéfinies déclenchent un comportement donné des services. La prise de décision du comportement à avoir est réalisée par un mécanisme de raisonnement sur le modèle du contexte. CASM et SOCAM ne traitent ni du chargement de services, ni de leurs dépendances. Hoareau [5] présente un déploiement distribué de composants, basé sur les contraintes dans des réseaux dynamiques, en considérant un modèle hiérarchique de composants qu'il déploie par propagation sur un ensemble de périphériques. La prise de décision des composants à déployer se fait grâce au descripteur de déploiement. Les dépendances entre composants sont des dépendances hiérarchiques statiques préétablies, alors qu'AxSeL propose une conception dynamique des dépendances.

3. AXSEL : UNE ARCHITECTURE DE CHARGEMENT CONTEXTUEL DE SERVICES

Nous présentons d'abord les paradigmes avec lesquels nous avons conçu notre système, soient le modèle de services, et le modèle de contexte.

3.1 Modèle de service

Plusieurs définitions du modèle composant ont été fournies [6]. Nous en retenons trois afin de positionner notre définition de service.

1. déf. 1 : Szyperski : *"A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."*
2. déf. 2 : Meyer : *"A component is a software element (modular element), satisfying the following conditions: (i) It can be used by other software elements, its "clients". (ii) It possesses an official usage description, which is sufficient for a client author to use it. (iii) It is not tied to any fixed set of clients.*
3. déf. 3 : Heineman and Councill : *"A software element that conforms to a component model and be independently deployed and composed without modification according to a composition standard".*

Nous définissons ainsi un service comme étant un objet fournissant une liste de fonctionnalités, et implanté par un composant qui intègre le code "métier" du service. Le composant implantant un service peut être déployé et exécuté (déf. 2 et déf. 1). Un composant peut publier plusieurs services. Un service n'est pas dédié à un utilisateur particulier (déf. 3). Nous attribuons également à nos services un modèle de contexte (cf 3.2). Notons deux points élémentaires dans un service :

- Les interfaces proposées : les services exportent des interfaces sur lesquelles des appels de méthodes peuvent être réalisés (déf. 1).
- Les interfaces requises : ce sont les interfaces utilisées par un service pour garantir son bon fonctionnement (déf. 1).

Un service est sujet à la composition avec d'autres services à travers les interfaces proposées/requises qu'il peut avoir (déf. 1). La figure 1 illustre le modèle de services considéré. Les dépendances entre les interfaces proposées et celles requises peuvent être satisfaites, s'il existe des services disponibles y répondant.

3.2 Modèle de contexte

"Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves."[2]

Nous attribuons à notre modèle de service/composant, un contexte (figure 2). Nous définissons le contexte comme étant l'ensemble des données pouvant être recueillies à partir des services (nom, version, besoin en ressources), des dispositifs (capacité d'accueil), et des utilisateurs (préférences).

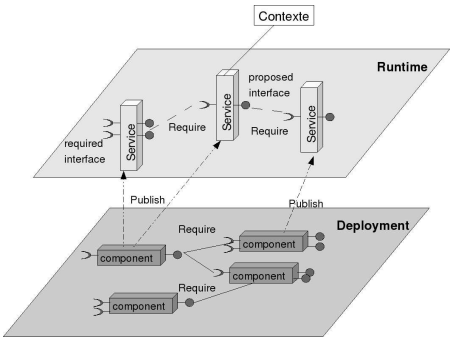

Figure 1: Modèle de service d'AxSeL

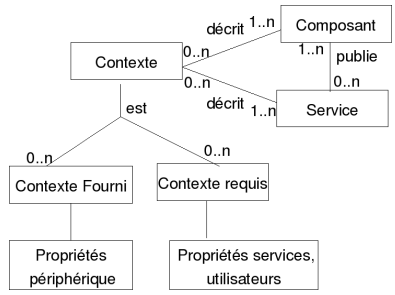

Figure 2: Liaison entre le modèle de services et le contexte

AxSeL utilise ces données du contexte pour effectuer le chargement des services, en confrontant un contexte requis (service, utilisateur) à un contexte fourni (noeud mobile). La comparaison permet de repérer les services à charger. Dans cet article, afin de simplifier la prise de décision, nous retenons uniquement deux données principales dans notre contexte:

- La priorité : est le degré d'importance d'un service par rapport à d'autres. Elle renseigne sur la nécessité de charger ou non un service. Si la priorité est haute le service est chargé, sinon il est moins utile de l'avoir sur le dispositif et nous gagnons ainsi l'espace de stockage qu'il aurait pu occuper.
- La taille mémoire : Celle-ci informe sur l'occupation d'espace mémoire d'un service donné. Si le service est plus grand que l'espace disponible sur le dispositif, nous ne pouvons pas le charger et optons, si possible, dans ce cas pour son remplacement par un autre service consommant moins de mémoire.

AxSeL peut considérer d'autres contraintes afin de mieux adhérer aux besoins des utilisateurs, dispositifs mobiles ou services. Ces contraintes peuvent relever des services (confiance, durée de vie, etc), du matériel (autres besoins en ressources matérielles), ou des préférences utilisateurs.

3.3 Architecture générale

La figure 3 présente une vue générale de l'architecture d'AxSeL. Les services sont hébergés dans des dépôts distants. Chaque dépôt est décrit par un descripteur. Les descripteurs de dépôts incluent des données sur les services (dépendances, emplacements, taille, etc.). Les noeuds mobiles accèdent à ces dépôts à travers leur descripteur de

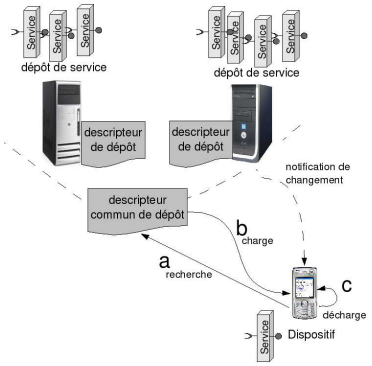

Figure 3: Comportement de la plate-forme AxSeL : (a) le dispositif recherche le service requis, (b) il le charge ainsi que ses dépendances localement, (c) en cas de changement le service peut être déchargé.

dépôt, et y puisent les services requis. Une fois trouvé, le composant implantant le service est chargé localement.

Le chargement se fait généralement sans prise en compte du contexte. AxSeL effectue cette prise en compte par une vue unifiée des différents dépôts de services au travers d'un unique descripteur.

Pour effectuer le chargement de services, AxSeL suit quatre étapes:

1. L'extraction des dépendances : est le processus par lequel les dépendances des services sont extraites à partir d'un descripteur de dépôt. Ce processus aboutit à un graphe de dépendances incluant les propriétés des services (graphe du descripteur commun de dépôt, figure 3).
2. La décision du chargement : cette étape consiste en un parcours du graphe de dépendances de services, en confrontant le contexte requis de ces services avec le contexte fourni des périphériques. Une décision de chargement ou non de service est prise (étape a, figure 3).
3. Le chargement : est l'étape qui applique la décision précédente en effectuant techniquement le téléchargement des composants publiant le service, son installation et son démarrage (étape b, figure 3).
4. L'adaptation contextuelle : est déclenchée soit par le changement d'état d'un ou plusieurs services, soit par la suppression des services jugés non nécessaires. Cette étape déclenche une nouvelle prise de décision avec comme conséquences d'éventuels chargements/déchargements (étape c, figure 3).

Dans les sections suivantes, nous présentons l'extraction des dépendances, la décision et l'adaptation contextuelle. Le mécanisme de chargement dépend de la plate-forme technique mise en place (délégation au chargeur de bundle Felix d'OSGi [1]).

3.3.1 *Extraction des dépendances : graphe bidimensionnel de services*

Les descripteurs de dépôts de services décrivent généralement un service, ses caractéristiques et ses

dépendances immédiates. Ainsi, ils ne permettent pas d'avoir une vue globale sur les services requis. Cette vue en même temps globale et dynamique, serait cependant, d'un grand intérêt vu qu'elle nous donnerait la possibilité de faire des choix sur les services intéressants à charger. Nous qualifions cette vue de dynamique car elle présentera les dépendances actuelles d'un service qui peuvent changer dans le temps, et ainsi modifier la vue.

Dans cette optique AxSeL fournit un modèle de description où nous définissons un service implanté par un ou plusieurs composants, et effectuant des compositions avec un ou plusieurs services. A partir de cette description, nous extrayons dans un graphe l'ensemble des dépendances possibles d'un service, ainsi que les données relatives aux services et aux composants (noms et emplacement des services/composants, taille mémoire requise). Nous incluons ces données dans les graphes extraits afin de les prendre en compte lors de la décision de chargement.

Nous considérons un graphe bidimensionnel (figure 4) incluant un niveau de services et un niveau de composants. Il s'agit d'un graphe orienté où les services et composants sont représentés par des noeuds et leurs dépendances par des arcs.

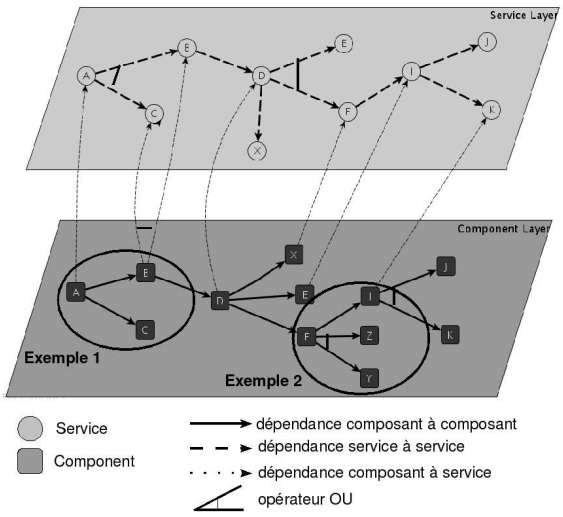

Figure 4: Graphe de dépendances de services

- Noeuds : les noeuds du graphe représentent les services et les composants. Les services sont représentés par des noeuds ronds alors que les composants par des noeuds carrés. Chaque noeud possède ses propres caractéristiques et son propre poids. Les noeuds services possèdent des poids relatifs aux services tels que (la priorité). Les noeuds composants ont des poids relatifs au niveau matériel (la taille mémoire).

- Arcs : représentent les dépendances. Nous distinguons entre dépendances d'exécution et dépendances de déploiement (Table 1). Les services sont implantés par des composants, ce lien est représenté par un arc vertical. Les composants importent d'autres composants, et les services importent d'autres services, ces dépendances sont représentées par des arcs horizontaux. Les arcs sont orientés et peuvent avoir des annotations spécifiques.

`Dépendances de déploiement`	Composant à service, composant à composant
`Dépendances d'exécution`	Service à service

Table 1: Classification des dépendances

Nous incluons dans notre modèle deux opérateurs logiques : ET et OU. Une dépendance portant l'opérateur ET renseigne sur la nécessité de charger les noeuds de cette dépendance. L'exemple 1 dans la figure 4 illustre ce cas. Le composant A doit charger les composants B ET C. Alors qu'une dépendance portant l'opérateur OU représente la possibilité de choisir un noeud ou un autre. Cet opérateur apparaît dans le cas de l'existence de plusieurs versions d'un service par exemple. Son inclusion dans le modèle est relative à la volonté de puiser les services de différentes sources. Dans l'exemple 2 de la figure 4, le composant F doit charger le composant I ET (Z OU Y).

3.3.2 Décision du chargement : coloration du graphe de services

Une fois le graphe de dépendances d'un service extrait, nous passons à l'étape suivante qui est la prise de décision des services à charger. Elle se base sur la confrontation de données contextuelles des services et des contraintes de la plate-forme. Ces dernières ne sont pas forcément compatibles, nous ne chargeons donc que les services obéissant à nos contraintes. La décision du chargement se base sur un algorithme de marquage par coloration des noeuds du graphe. Nous parcourons le graphe bidimensionnel et affectons à ses noeuds une de ces deux couleurs :

- Le rouge est affecté aux services obéissant aux critères de chargement et qui seront chargés localement sur la plate-forme.

- Le blanc est affecté aux services n'obéissant pas aux contraintes prédéfinies. Dans ce cas la décision est la création et la re-direction vers un service nul. Notons que la couleur blanche ne peut pas briser une chaîne de noeuds rouges.

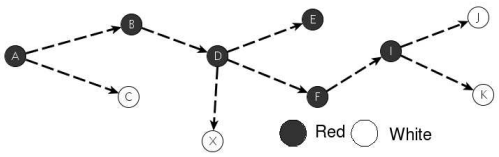

Figure 5: Coloration de graphes de services

Lors du parcours du graphe (figure 5), lorsqu'un noeud rouge est rencontré, il est chargé. Lorsqu'il s'agit d'un noeud blanc, AxSeL crée et redirige vers un service nul. Nous sommes conscients que rediriger vers un service nul, ne fait pas pleinement fonctionner l'application chargée. Mais,

cependant, de cette manière nous garantissons la bonne exécution d'une application (pas de crash dû à un service manquant) et de ne charger que le nécessaire, en attendant la présence d'un autre service équivalent et plus adéquat.

3.3.3 Adaptation dynamique : déclenchement événementiel

AxSeL prend en compte la dynamique du contexte. Cette dynamique est mise en place à l'aide d'une approche événementielle. Notons deux types d'événements : d'abord, les évènements liés au changement du dépôt de services : des nouveaux services apparaissent, ou des services déjà chargés sont mis à jour (nouvelle version, nouvelle implantation, indisponibilité). Ensuite, les évènements liés au périphérique : ressource mémoire insuffisante ou libérée, changement de priorités. Pour répondre à ces événements, deux types d'actions sont possibles sur le graphe :

- Action de modification de la structure du graphe : ajout/retrait de noeud, dépendances, etc.
- Action de changement de la coloration d'un noeud : mémoire libérée/occupée, choix utilisateur, changement de l'état d'un service, etc.

Les actions listées entrainent une nouvelle prise de décision. L'algorithme de coloration considéré, est incrémental et prend en compte ce qui a déjà été chargé, pour ne pas reparcourir le graphe et le re-colorier.

4. RÉALISATION D'AXSEL

La figure 6 présente l'architecture en couches de la plate-forme d'exécution. Le prototype d'AxSeL est réalisé en utilisant le langage Java et la plate-forme orientée services felix OSGi. Nous faisons correspondre notre modèle de services

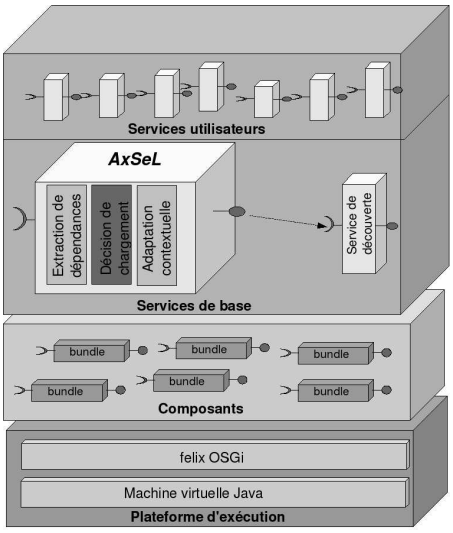

Figure 6: Vue générale de la plate-forme d'exécution

aux services OSGi et les composants aux bundles OSGi. AxSeL est un service appartenant aux services de base. Il peut utiliser des services tels que le service de découverte qui permet de capter les événements contextuels. AxSeL contient trois parties essentielles : l'extraction de dépendances (cf 3.3.1), la décision du chargement (cf 3.3.2) et l'adaptation contextuelle (cf 3.3.3).

4.1 Algorithme d'extraction du graphe

Nous partons du descripteur de dépôt de composants OBR d'OSGi, que nous enrichissons par une couche service pour qu'il corresponde à notre modèle de services. Ensuite, nous exploitons l'API OBR fournie pour rechercher les dépendances d'un service donné. Enfin, nous en construisons un graphe bidimensionnel de services/composants (cf 3.3.1) avec l'algorithme 1 :

Algorithm 1 Graph extractDependency(Resource resource)

```
1: G ← new Graph(resource)
2: Requirement children[ ] ← resource.getRequirements()
3: ∀ i Gi ← children extractDependency(children[i])
4: if Gi.entryNode provides Service then
5:    include Gi.entryNode in ServiceLayer
6: else if Gi.entryNode provides Component then
7:    include Gi.entry in ComponentLayer
8: end if
9: G.include(Gi)
10: G.addDependencyEdge(G.entryNode, Gi.entryNode)
11: return (G)
```

Le processus se fait sur deux opérations :

1. Retrouver les dépendances : cette opération se base sur l'analyse du descripteur du dépôt. Pour chaque service/composant (Resource) nous retrouvons récursivement l'ensemble de ses dépendances (Requirement) en termes de services et composants. Nous parcourons aussi les éléments dépendant pour en dégager les dépendances.
2. Construire le graphe : cette opération est réalisée au fur et à mesure que l'opération précédente se déroule. Les noeuds sont créés et inclus dans le graphe selon leur type. Un service correspondra à un noeud appartenant au niveau des services et un composant correspondra à un noeud du niveau des composants. Les liens sont inclus entre les noeuds d'un même niveau et ceux de différents niveaux. A chaque étape de l'algorithme, l'inclusion entre le graphe courant et le graphe de dépendances de ses fils est enrichi d'une dépendance entre le point d'entrée du graphe courant et ses fils.

4.2 Algorithme de coloration de graphe

L'algorithme de coloration utilise deux métriques :

- d(sol) représente la somme des tailles mémoires des services (noeuds) de la solution en cours de coloration. d(Max) est une valeur constante représentant la taille mémoire maximale allouée sur un dispositif. Au delà de cette valeur nous pouvons plus charger de services.
- prio(N) est une fonction qui retourne la priorité d'un service. prio(Min) est le niveau en dessous duquel les services ne pourront pas être choisi.

La coloration de graphe s'effectue en deux étapes:

1. Coloration initiale en blanc : l'ensemble du graphe est coloré en blanc au départ. Par défaut, aucun service n'est donc chargé mais redirigé vers un service nul. Cette étape est absolument non coûteuse car nous ne

parcourons pas réellement le graphe pour le colorier, mais nous avons prémarqué les noeuds en blanc lors de leur création.

2. Coloration en rouge : nous parcourons réellement le graphe pour effectuer la coloration en rouge en décidant à chaque noeud s'il respecte les contraintes du contexte requis par rapport à celles du contexte fourni par le périphérique. Cette étape s'initie en appelant l'algorithme 2 avec le point d'entrée du graphe de dépendances et une valeur de solution courante nulle : colour(new List[G.entryNode], 0)

Algorithm 2 (List, d(sol)) colour(List l, Integer d(sol))

```
1: l.sort(prio(node))
2: firstNode = l.firstElement()
3: if (prio(firstNode) ≥ prio(Min)) and (d(sol) + firstNode.size ≤ d(Max)) then
4:    firstNode.changeColour(red)
5:    return colour(l - firstNode + children(firstNode), d(sol) + firstNode.size)
6: else
7:    return colour(l - firstNode, d(sol))
8: end if
```

L'algorithme trie les noeuds selon leur priorité, ensuite prend le premier noeud de la liste triée. Si ce noeud obéit aux contraintes de priorité et de taille mémoire il est coloré en rouge, et nous procédons au traitement de ses fils. L'algorithme retourne une liste de noeuds restant à traiter et la taille de la solution courante.

4.3 Algorithme d'adaptation

L'algorithme d'adaptation du chargement (algorithme 3) est appelé sur notification d'un événement de l'environnement auquel AxSeL est abonné. Les événements listés peuvent venir de deux sources :

- Ajout d'un noeud : nous vérifions s'il a un parent déjà chargé. Dans ce cas nous retraitons les noeuds avec le nouveau noeud, et la taille mémoire courante libre.
- Changement de l'état d'un noeud : Si le noeud est déjà chargé nous le déchargeons, et libérons la mémoire qu'il occupait. Nous traitons également ses fils (algorithme 4).

La fonction uncolor(List) permet de vérifier qu'un noeud à décharger n'a pas de prédécesseurs chargés. La taille du noeud est déduite de la mémoire et le traitement est propagé aux fils.

5. CONCLUSION

Nous avons proposé, dans cet article, AxSeL : une architecture adaptative de chargement contextualisé de services. Nous y proposons également un nouveau modèle de service représenté sous la forme d'un graphe bidimensionnel de services/composants. En considérant ce graphe, nous prenons la décision de charger ou non les services. Le processus de prise de décision est réalisé par un algorithme de coloration de graphe. La coloration se base sur des contraintes contextuelles issues des services et des plates-formes mobiles. Le prototype d'AxSeL est actuellement en cours de test afin d'évaluer ses performances et de le comparer à des approches

Algorithm 3 adaptation(Event e)

```
1: node = e.getSource()
2: if (e.getType() == NewNode) then
3:    if ∃ nodei tq father(node) and nodei.isColored(red) then
4:       colour(l + node, d(sol))
5:    end if
6: else if (e.getType() == NodeChange) then
7:    if (node.isColored(red)) then
8:       node.changeColour(white)
9:       d(sol) ← d(sol) - node.size
10:      uncolor(new List (children(node)))
11:      color(l + node, d(sol))
12:   else if ((node.isColored(white)) and (∃ nodei tq father(node)) and (nodei.isColored(red)) then
13:      colour(l + node, d(sol))
14:   end if
15: end if
```

Algorithm 4 uncolor(List l)

```
1: ∀ node ∈ l
2: if (node.isColored(red) and @ nodei tq father(node) and nodei.isColored(red)) then
3:    node.changeColor(white)
4:    d(sol) ← d(sol) - node.size
5:    uncolor(l - node + children(node))
6: end if
```

similaires. Comme travaux futurs, nous prévoyons à court terme de tester les performances d'AxSeL sur l'ensemble de services existants dans le dépôt de felix (environ 350 bundles). A long terme, nous intégrerons également la notion de services distants trop volumineux pour être chargés localement.

6. REFERENCES

[1] The Apache Felix Project. The Apache Felix Project http://cwiki.apache.org/FELIX/index.html, 2008.

[2] Dey, A., Salber, D., and Abowd, G. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. In *HCI conference* (2001).

[3] Gu, T., Pung, H., and Zhang, D. A Middleware for Building Context-Aware Mobile Services. In *IEEE VT Conference* (2004).

[4] Hall, R. S., and Cervantes, H. An OSGi Implementation and Experience Report. In *IEEE CCNC conference* (2004).

[5] Hoareau, D., and Mahéo, Y. Constraint-Based Deployment of Distributed Components in a Dynamic Network. In *ARCS conference* (2006).

[6] K.Kui, and Z.Wang. Software component models. *IEEE TSE conference* (2007).

[7] OBR. Obr Bundle Repository. http://oscar-osgi.sourceforge.net/, 2006.

[8] Park, N., Lee, K., and Kim, H. A Middleware for Supporting Context-Aware Services in Mobile and Ubiquitous Environment. In *IEEE ICMB conference* (2005).

[9] Poladian, V., Sousa, J., Garlan, D., and Shaw, M. Dynamic configuration of resource-aware services. In *26th ICSE conference* (2004).